

AD-A049 475

TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF
SEMANOL(76) INTERPRETER DOCUMENTATION. VOLUME IV.(U)
NOV 77 E R ANDERSON, D M HEIMBIGNER

F/G 9/2

F30602-76-C-0238

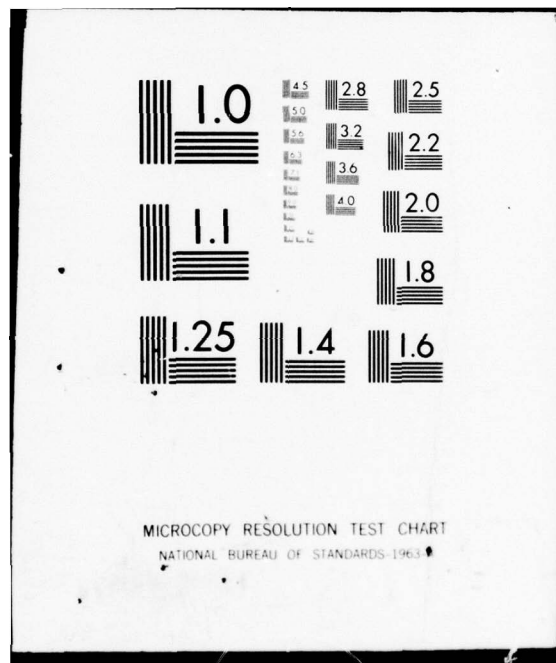
UNCLASSIFIED

RADC-TR-77-365-VOL-4

NL

1 of 1
AD
A049475





AD A 049475

RADC-TR-77-365, Vol IV (of four)
Final Technical Report
November 1977

2



SEMANOL(76) INTERPRETER DOCUMENTATION

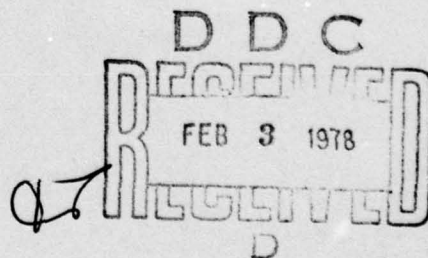
Eric R. Anderson
Dennis M. Heimbigner

TRW Defense and Space Systems Group

AD No. _____
RADC FILE COPY

Approved for public release; distribution unlimited.


ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441



This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

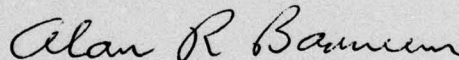
RADC-TR-77-365, Vol IV (of four) has been reviewed and approved for publication.

APPROVED:



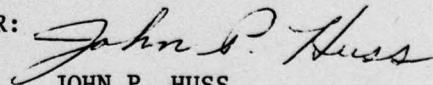
JOHN M. IVES, Captain, USAF
Project Engineer

APPROVED:



ALAN R. BARNUM, Assistant Chief
Information Sciences Division

FOR THE COMMANDER:



JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

*MISSION
of
Rome Air Development Center*

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C³), activities, and in the C³ areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.



UNCLASSIFIED

TR-77-365-VOL-4

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADG-TR-77-365, Vol IV (of four)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SEMANOL(76) INTERPRETER DOCUMENTATION, Volume IV.	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report	
6. AUTHOR(s) Eric R./Anderson Dennis M./Heimbigner	7. PERFORMING ORG. REPORT NUMBER N/A	
8. PERFORMING ORGANIZATION NAME AND ADDRESS TRW Defense and Space Systems Group One Space Park Redondo Beach CA 90278	9. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0238 new	
10. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	11. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 63728F J.O. 5550840	
12. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	13. REPORT DATE Nov 77	
14. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.	15. NUMBER OF PAGES 32	
15. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same	16. SECURITY CLASS. (of this report) UNCLASSIFIED	
16. SUPPLEMENTARY NOTES RADG Project Engineer: Captain John M. Ives (ISIS)	17. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
18. KEY WORDS (Continue on reverse side if necessary and identify by block number) SEMANOL standardization SEMANOL(76) language control Semantics metalanguage processor syntax interpreter Language definition		
19. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report describes the internal organization of the SEMANOL(76) interpreter program, operational upon the Multics computer system, and provides instructions on the use of the program. The SEMANOL(76) interpreter program is a processor of metaprograms written in the SEMANOL(76) metalanguage (i.e., it executes SEMANOL(76) metaprograms); these metaprograms are ordinarily formal definitions of object programming languages.		

DD FORM 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409 637

Gue

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

REPORT DOCUMENTATION PAGE	
Form 101 (Rev. 10-1-75)	
1. REPORT NUMBER	
2. AUTHOR	
3. TITLE	
4. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	
5. PERFORMING ORGANIZATION REPORT NUMBER	
6. AUTHORING OR PERFORMING ORGANIZATION	
7. AUTHOR	
8. PERFORMING ORGANIZATION	
9. PERFORMING ORGANIZATION	
10. PERFORMING ORGANIZATION	
11. PERFORMING ORGANIZATION	
12. PERFORMING ORGANIZATION	
13. PERFORMING ORGANIZATION	
14. PERFORMING ORGANIZATION	
15. PERFORMING ORGANIZATION	
16. PERFORMING ORGANIZATION	
17. PERFORMING ORGANIZATION	
18. PERFORMING ORGANIZATION	
19. PERFORMING ORGANIZATION	
20. PERFORMING ORGANIZATION	
21. PERFORMING ORGANIZATION	
22. PERFORMING ORGANIZATION	
23. PERFORMING ORGANIZATION	
24. PERFORMING ORGANIZATION	
25. PERFORMING ORGANIZATION	
26. PERFORMING ORGANIZATION	
27. PERFORMING ORGANIZATION	
28. PERFORMING ORGANIZATION	
29. PERFORMING ORGANIZATION	
30. PERFORMING ORGANIZATION	
31. PERFORMING ORGANIZATION	
32. PERFORMING ORGANIZATION	
33. PERFORMING ORGANIZATION	
34. PERFORMING ORGANIZATION	
35. PERFORMING ORGANIZATION	
36. PERFORMING ORGANIZATION	
37. PERFORMING ORGANIZATION	
38. PERFORMING ORGANIZATION	
39. PERFORMING ORGANIZATION	
40. PERFORMING ORGANIZATION	
41. PERFORMING ORGANIZATION	
42. PERFORMING ORGANIZATION	
43. PERFORMING ORGANIZATION	
44. PERFORMING ORGANIZATION	
45. PERFORMING ORGANIZATION	
46. PERFORMING ORGANIZATION	
47. PERFORMING ORGANIZATION	
48. PERFORMING ORGANIZATION	
49. PERFORMING ORGANIZATION	
50. PERFORMING ORGANIZATION	
51. PERFORMING ORGANIZATION	
52. PERFORMING ORGANIZATION	
53. PERFORMING ORGANIZATION	
54. PERFORMING ORGANIZATION	
55. PERFORMING ORGANIZATION	
56. PERFORMING ORGANIZATION	
57. PERFORMING ORGANIZATION	
58. PERFORMING ORGANIZATION	
59. PERFORMING ORGANIZATION	
60. PERFORMING ORGANIZATION	
61. PERFORMING ORGANIZATION	
62. PERFORMING ORGANIZATION	
63. PERFORMING ORGANIZATION	
64. PERFORMING ORGANIZATION	
65. PERFORMING ORGANIZATION	
66. PERFORMING ORGANIZATION	
67. PERFORMING ORGANIZATION	
68. PERFORMING ORGANIZATION	
69. PERFORMING ORGANIZATION	
70. PERFORMING ORGANIZATION	
71. PERFORMING ORGANIZATION	
72. PERFORMING ORGANIZATION	
73. PERFORMING ORGANIZATION	
74. PERFORMING ORGANIZATION	
75. PERFORMING ORGANIZATION	
76. PERFORMING ORGANIZATION	
77. PERFORMING ORGANIZATION	
78. PERFORMING ORGANIZATION	
79. PERFORMING ORGANIZATION	
80. PERFORMING ORGANIZATION	
81. PERFORMING ORGANIZATION	
82. PERFORMING ORGANIZATION	
83. PERFORMING ORGANIZATION	
84. PERFORMING ORGANIZATION	
85. PERFORMING ORGANIZATION	
86. PERFORMING ORGANIZATION	
87. PERFORMING ORGANIZATION	
88. PERFORMING ORGANIZATION	
89. PERFORMING ORGANIZATION	
90. PERFORMING ORGANIZATION	
91. PERFORMING ORGANIZATION	
92. PERFORMING ORGANIZATION	
93. PERFORMING ORGANIZATION	
94. PERFORMING ORGANIZATION	
95. PERFORMING ORGANIZATION	
96. PERFORMING ORGANIZATION	
97. PERFORMING ORGANIZATION	
98. PERFORMING ORGANIZATION	
99. PERFORMING ORGANIZATION	
100. PERFORMING ORGANIZATION	

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

ACCESSION FOR	
NTIS	Write Section <input checked="" type="checkbox"/>
GDS	Both Sections <input type="checkbox"/>
UNANNOUNCED	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
DCL. AVAIL. and/or SPECIAL	
A	

ABSTRACT

The SEMANOL(76) Interpreter computer program is a processor of metaprograms written in the SEMANOL(76) metalanguage; that is, the Interpreter executes SEMANOL(76) statements. Since SEMANOL(76) is a metalanguage for syntactic and semantic specification, the metaprograms processed by the Interpreter are ordinarily formal definitions of object programming languages. The SEMANOL(76) Interpreter is operational upon the Multics MR 5.0 system.

CONTENTS

	<u>Page</u>
1. Program Name	1
2. Authorship	1
3. Support System Definition	1
4. Program Description	1
4.1 The Translator	2
4.2 The Executer	5
5. Logic Diagrams	14
6. Inputs	14
6.1 Translator Input	14
6.2 Executer Input	14
7. Output	14
7.1 Translator Outputs	14
7.2 Executer Outputs	14
8. Program Setup	18
9. Operating Instructions	18
9.1 Translator Command	18
9.2 Executer Commands	20

1. Program Name

The name of this total computer program is the "SEMANOL(76) Interpreter". The SEMANOL(76) Interpreter is then made up of two major subprograms called the "Translator" and the "Executer".

2. Authorship

The SEMANOL(76) Interpreter program was written by TRW Defense and Space Systems Group under contract F30602-76-C-0238. The responsible programmers were Eric R. Anderson, telephone (213) 536-3217, and Dennis M. Heimbigner, telephone (213) 536-2914. The sponsoring agency was the Rome Air Development Center; Captain John M. Ives was project engineer.

3. Support System Definitions

The SEMANOL(76) Interpreter runs on a HIS-6180 computer under control of the Multics MR 5.0 operating system. Its core memory requirement depends mainly upon the size of the SEMANOL(76) metalanguage program being run. The Interpreter is normally operated in a timesharing mode, and thus requires that file space be available on the permanent file disk unit. The Interpreter does not use magnetic tape, card devices, nor the on-line printer.

4. Program Description

The SEMANOL(76) Interpreter has been implemented through the use of two major subprograms as observed earlier. The first subprogram, the Translator, reads a SEMANOL(76) program describing a programming language and converts it to a form (called SIL) which is readily usable by the next subprogram, the Executer. The Executer loads and executes this SIL. The Executer actually consists of a number of programs which communicate through a COMMON data base. Its commands include initialization and running commands, breaking commands, syntactic component commands, tracing commands, and miscellaneous commands. All of these features provide the user with great flexibility when executing SEMANOL(76) metaprograms.

The SIL file used for communication between the Translator and Executer is described with the Executer. It is an alphameric representation of a list of operands and operators produced by processing the SEMANOL(76) program. Note that these subprograms were written in Fortran, apart from some PL/1 routines which were needed for special functions (e.g., doing half word load and stores and input/output). Each subprogram is described separately in what follows, and indeed they are rather independent due to the minimum interface resulting from the use of the SIL file. Please note that the program listings are richly annotated and contain the full details of program implementation.

4.1 The Translator

The SEMANOL(76) Translator translates a SEMANOL(76) language source program into a SEMANOL interpreter language (SIL) object program. The translator uses the recursive-descent method to analyze the syntax of the source program. Recursive descent is a top-down, predictive recognition process employing one recursive procedure for each of the syntax rules of the source language. This method was chosen because it allows the construction of a modular translator program; changes or additions to the source language syntax are easily accommodated since there is little interdependence among the recognition procedures.

The translator contains the following functional groups.

- A. Initialization
- B. Lexical analysis
- C. Syntactic analysis (Parsing)
- D. SIL code generation and output
- E. Error handling and final report.

It is assumed that some command interface exists, which is operating system dependent. This command interface opens the SEMANOL(76) source program, the SIL output program, and collects parameters which define the translation options. This command interface invokes the major FORTRAN subroutine, TRANS and passes the option parameters. TRANS invokes in succession INIT, PARSE, and REPORT.

The major initialization procedure, INIT, is responsible for initializing the translator data structures. The major data structures are:

- A. String space
- B. Keyword hash table
- C. SIL output tokens
- D. Symbol table
- E. SIL list space

The lexical analyzer scans the source language program, one line at a time, and converts each line to an equivalent array of tokens. A token is a data structure which represents information about a substring of the input line. In some cases, it may even contain the substring. Often the word "token" is used to denote the substring as well as the data structure. In particular, the tokens of the SEMANOL(76) Translator define:

- A. Type, which is one of
 - 1. keyword
 - 2. delimiter
 - 3. string constant
 - 4. integer constant
 - 5. bit-string constant
 - 6. name
 - 7. end-of-file
 - 8. illegal-token

B. Value, whose interpretation depends on the type

1. keyword - value is an index to the keyword string in the string-space.
2. delimiter - value is not used.
3. string, integer, or bit-string constant - value is an index to the text of the constant.
4. name - value is a symbol table index.
5. end-of-file, illegal - value is not used.

C. Mark, which is a character index into the source line containing the first character of the token.

Blanks, comments, and ends-of-line are not themselves tokens. They only delimit tokens in the input line. Because ~~end-of-line~~ is not part of any token, no token may extend across lines, hence each line must contain an integral number of tokens.

The SEMANOL(76) Translator uses a recursive descent parsing scheme. There is no backtrack, but there is a one token look-ahead capability. Each subroutine of the parser represents a non-terminal of the SEMANOL(76) grammar. However, not all non-terminals are represented by subroutines. Each subroutine is responsible for any necessary code generation relating to its non-terminal. Some subroutines do not generate code, generally because they are defined in terms of other non-terminals. In general, the recursive descent subroutines define a compressed form of the SEMANOL(76) grammar.

A typical parse subroutine attempts to form an instance of its defining syntax rule by inputting tokens when it expects some terminal (such as a keyword) and recursively calling a parse subroutine when it expects a non-terminal. Failure to match an expected grammar item will result in a fail return of that parse subroutine.

The ultimate aim of the Translator is to output the strings of SIL code which represent the SEMANOL(76) source program being translated. Code generation has two aspects. One aspect is the code which is generated corresponding to a SEMANOL(76) construct. The second aspect is the procedures available to generate code.

As the source language statements are parsed, the corresponding SIL translation is generated by calls to the SIL generator procedures. These subroutines allow generation of lists of SIL code operations, constants, labels, and strings. After each #DF or #PROC-DF is parsed, the lists of SIL code are converted to strings and output to the SIL program file.

The generated SIL code is stored in a list structure. Because the SIL grammar order is different from SEMANOL(76), procedures are available to utilize multiple lists and to merge lists in an arbitrary order.

Whenever the Translator discovers an error, one of four procedures is called. Each procedure handles different kinds of errors.

- LEXERR - lexical analysis errors
- SYNERR - syntactic and semantic analysis errors
- WERR - non-fatal warnings
- CERR - errors resulting from failure of the translator, such as table space overflow.

Each error routine is passed an explanatory message. The error routine combines this message with other information. There are six pieces of information output for each error.

1. SECTION - section of the SEMANOL(76) source program containing the error
2. DF - the name of the syntactic or semantic #DF containing the error.
3. LINE NUMBER - line number (beginning with one) of the line containing the error.
4. TEXT - the text of the line containing the error.
5. MARKER - an indicator of the point in the line where the error occurred.
6. MESSAGE - the error message as passed to the error routine.

If any of the first five pieces has already been output and has not changed, then it is not output again. CERR never outputs the marker since it is generally meaningless when the Translator fails.

After completion of the parse and code generation for the whole input source file, the Translator top level procedure calls REPORT. REPORT produces more error information and summary statistics.

REPORT scans the final symbol table to detect and report four kinds of errors.

1. The names of all syntactic and semantic #DF's which contained an error.
2. All global, syntactic, and semantic names which were referenced but not defined.
3. The number of names with no defined type.
4. All global, syntactic, and semantic names which were defined but not referenced.

Finally, the report routine outputs various statistics, such as space used and total lines read.

4.2 The Executer

The Executer is that part of the SEMANOL(76) Interpreter which actually executes SEMANOL(76) metaprograms. Each Executer command is a separate program called from Multics command level. The programs communicate through Fortran COMMON blocks which are initialized by the `semanol` command. Note that the Translator does not reference these COMMON blocks; it only communicates with the Executer commands through SIL files.

The first step prior to using the Executer is to establish links to the 21 Executer commands. After this, the first command executed must be the `semanol` command. The command

`semanol pathname`

accomplishes several things:

1. It initializes the Executer and the COMMON blocks which provide communication between the commands.
2. It loads all of the #DFs on the ASCII text segment given by `pathname` (presumably this segment contains the SIL output of a previous translate command).

As soon as the `semanol` command is given, the INIT subroutine is called. The purpose of INIT is to initialize all of the variables and tables used in COMMON by the Executer. These structures are described in the following paragraphs in the order in which they are initialized:

1. The descriptor table is zeroed except for the ITL field of each entry. The free entry list is constructed using this field as a pointer.
2. The string array, ICHAR, is zeroed.
3. The main stack is zeroed.
4. The symbol table bucket array, IBUCK, is zeroed.
5. The SIL code array, ICODE, is zeroed.
6. Simple global variables are set to their initial values.
7. The global variables OP, VAR, SYN, VAL, and SIL are initialized to contain their associated attribute type numbers.
8. The constant list is initialized.
9. The null sequence, #UNDEFINED, #TRUE, #FALSE, and the null string, #B2, and #B8 are put on the constant list.
10. Op-code values are initialized by calling the INOP subroutine.
11. Finally, INIT returns.

While initialization is going on, many of the general utility routines are called. A list of some of these utility routines and what they do follows:

1. The PL/1 functions ITYPE, IHDA, IHD, ITL, ICT and IPTR are called with a descriptor table or stack index and return the value of the named field at that index.

2. The PL/1 subroutines STYPE, SHDA, SHD, STL, SCT, and SPTR do the inverse; they store a value into the associated field of the descriptor table or stack.
3. LITSTR is used to create a literal string and push its descriptor onto the stack.
4. LKPN looks up a name in the symbol table.
5. PUSH pushes a value onto the main stack.

These are just a few of the utilities used by the initialization, but they give an idea of the kinds of operations required.

When initialization is complete, the SIL program is read from the segment specified by the `semanol` command argument. The JSILSM routine parses each SIL statement and converts it to internal form. Semantic definitions and the special SIL statement (`#CONTROL/SIL`) corresponding to the `#CONTROL-COMMANDS` section are stored in the ICODE array. Syntactic definitions are stored as lists in the main descriptor table.

The `run` command is used to actually begin execution of a `SEMANOL(76)` metaprogram. Assume that the `semanol` command has just been executed, initializing the COMMON areas and loading a SIL program. Then the `run` command is used to start a test case.

INTERP is the main run subroutine. It calls the operator subroutines as required by the SIL code. Since the flow of control is directed by INTERP, which is in turn directed by the SIL metaprogram, an understanding of SIL is essential to an understanding of the Executer.

The external syntax of SIL is extremely simple (See Table I). Although the SIL program is not stored internally in string form, it will be assumed that the INTERP subroutine works directly on the string format as this assumption elucidates the following discussion.

A SIL program consists of two kinds of statements, syntax statements and semantic statements. An example of a syntax statement is

```
PROGRAM/SYN = [ ( SCAT/OP STMT/SYN ( KSTAR/OP
                  STMT/SYN KEND/OP ) ) ] ;
```

An example of a semantic statement is

```
PRINTA/SIL = [ 'A' OIODAT/OP MSTOP/OP ] ;
```

The difference between the two is that a syntax statement always has statement-attribute 'SYN' and corresponds to a `SEMANOL(76)` syntactic `#DF`. The syntax statements are not executed directly by INTERP, but instead are used by the parsing subroutine JPARSE. In contrast, semantic statements have statement-attribute 'SIL'. They are read directly by INTERP which calls the operator subroutines to execute them.

The syntax statements will be discussed with the parser. Semantic statements can be divided into two types as identified by their statement-name. Corresponding to the SEMANOL(76) program in the #CONTROL-COMMANDS section is the statement with statement-name '#CONTROL'. All other SIL semantic statements correspond to semantic #DFs and #PROC-DFs. The subroutine INTERP treats all of these routines the same once execution begins.

The elements within a SIL semantic statement list are normally processed from left to right like a reverse Polish string of operators and operands. Operands and results of operations are kept on a stack. The actions taken when each kind of element is encountered are summarized below:

1. If a <name> < '/' > <'SIL','SYN'> or <name> < '/' > <'VAL'> is encountered, a pointer to the symbol table entry for the given element is pushed onto the stack. Note that each <name> < '/' > <attribute> has its own symbol table entry.
2. If <integer> < '/' > <'VAR'> is encountered, a pointer to the stack entry for the parameter or local variable is pushed onto the stack.
3. If a constant is encountered, its value is pushed onto the stack.
4. If a <name> < '/' > <'OP'> is encountered, the operator with the given name is executed.

At this point, consider case 4 above. The INTERP subroutine handles each operator by calling a subroutine whose name is the same as the operator subroutine name. The operators which require operands take them from the top of the stack. They often replace their operands with a result.

The operands for functional operators are descriptors for, or pointers to, the various SIL data types. The data types used are:

- | | |
|-----------------------|----------------------------|
| 1. UND - #UNDEFINED | 4. SEQ - A finite sequence |
| 2. IINT - An integer | 5. STR - A string |
| 3. PRS - A parse tree | 6. LOG - #TRUE or #FALSE |

The stack contains pointers to symbol table entries and it also contains special entries marking #DF and #PROC-DF calls.

Consider the following segment of SIL code:

```
#I 1 #I 2 A/VAL CVS/OP STLFT/OP STRIT/OP A/VAL ASVAR/OP
```

Suppose the Executer encountered this code as an element of a routine. It would first push the integers 1 and 2 onto the stack. When it encountered A/VAL it would push a pointer to the symbol table entry for the global A/VAL onto the stack. But then CVS/OP (convert to string operator) would go to the A/VAL symbol table entry, get the value stored there, convert it to a string, and push the string descriptor onto the stack in place of the pointer to A/VAL. The top of the stack (at right) would now contain

```
...1,2,'ABC'
```

given that A/VAL had a value of 'ABC'. The Executer would then call the STLFT subroutine which would replace 2 and 'ABC' with 'AB', implementing the SEMANOL(76) #LEFT 2 #CHARACTERS-OF A. Next, the STRIT subroutine would be called to replace 1 and 'AB' with 'B', implementing the SEMANOL(76) #RIGHT 1 #CHARACTERS-OF (#LEFT 2 #CHARACTERS-OF A). The top of the stack would now contain

```
... 'B'
```


Next, another pointer to the A/VAL symbol table entry would be pushed on the stack and the ASVAR subroutine would be called to store 'B' at the A/VAL symbol table entry location. ASVAR deletes its arguments without leaving anything on the stack, so now the stack would be as it was at the start and the code would be complete.

Not all operators pass control in the sequential manner illustrated. The operator BRANCH is an unconditional branching operator which is immediately followed, as are other branching operators, with a relative branch address. Consider the following (ludicrous) SIL code:

BRANCH/OP 2 MSTOP/OP MERRS/OP 3

When control reaches the BRANCH/OP, the relative branch address (2) is processed. The relative count is from the 2 itself, and a positive number indicates forward branching; hence, MERRS/OP is the next operator executed. The MSTOP/OP is skipped. Had the branch address been negative, the branch would have been a backward one.

There are seven conditional branching operators and they are BDEC, BEMPTY, BFALSE, BINTSD, BINTST, BSETST, and BTRUE. In SIL code, each of them is followed by a relative branch address; however, the branch is taken depending on some kind of test. If the branch is not taken, control next goes to the point immediately after the operator and its branch address. As an example, consider the following SIL code:

B/VAL CVS/OP 'A' PEQW/OP BTRUE/OP 2 MSTOP/OP...

Suppose control has come to the B/VAL element. A pointer to the B/VAL symbol table entry is pushed onto the stack. The CVS subroutine converts this pointer to the string value of the variable B. 'A' is then pushed onto the stack. Now the PEQW subroutine compares the top two stack entries (assuming they are strings). If they are identical it replaces them with #TRUE and if not it replaces them with #FALSE. (This implements the SEMANOL(76) arg1 #EQW arg2.) Now, the BTRUE operator is executed. If the top stack entry is #TRUE it branches. Otherwise, BTRUE drops through and control goes to the MSTOP operator.

Other operators can interrupt normal left to right processing within a sublist. They include the following:

DCALL/OP - DCALL is used to implement a SEMANOL(76) #DF or #PROC-DF call. When it is encountered in a code list, the top of the stack contains (starting at the top) a pointer to the symbol table entry containing the code for the #DF to be called and the argument values themselves. An integer after the DCALL operator indicates the number of arguments being passed.

RET/OP - - RET implements a SEMANOL(76) #DF return. When it is encountered, control returns to the point at which the last DCALL/OP was executed. The value returned is the one at the top of the stack when RET/OP is executed.

Now, an example of a SEMANOL(76) semantic #DF is given, showing how it is represented as an SIL statement, and then following INTERP through its interpretation. This will illustrate the #DF calling mechanism of the Executer. The SEMANOL(76) #DF follows:

```
#DF LONGER (STRINGA, STRINGB)
```

```
=> STRINGA #IF #LENGTH (STRINGA) >= #LENGTH (STRINGB);
=> STRINGB #OTHERWISE #.
```

This #DF returns the longer of its two string arguments. It translates into the SIL statement

```
LONGER/SIL =
```

```
(1) [ 2 2 2
(2) 1/VAR CVSTS/OP ISLEN/OP CVI/OP 2/VAR
(3) CVSTS/OP ISLEN/OP CVI/OP PLT/OP LNOT/OP
(4) BFALSE/OP 3 1/VAR RET/OP
(5) 2/VAR RET/OP ] ;
```

(Note that the parenthesized numbers are for reference purposes and do not appear in the SIL code.) Suppose for the illustration that the arguments are STRINGA-'AB' and STRINGB-'ABC' when the #DF is called. The Executer finds itself at line (1) in the above code. This line contains three integers similar to those found at the head of the SIL code for any #DF. The first integer indicates the default values for trace, syntactic component, and break flags. The second integer indicates how many parameters the #DF expects (in this case 2). The third integer indicates how many stack entries must be allocated to parameters and local variables for this #DF (again, in this case 2). Lines (2) and (3) indicate that the Executer is to calculate and compare the lengths of the two strings. 1/VAR references the first parameter (STRINGA) and 2/VAR references the second parameter (STRINGB). Since STRINGA is shorter, #FALSE will be left on the stack after execution of lines (2) and (3). At the start of line (4) is BFALSE/OP 3. Since this branches if false and #FALSE is encountered on top of the stack in this case, control skips over the rest of line (4), directly to line (5). (The relative branch is 3 forward from the 3, or right to the beginning of line (5).) Finally, line (5) indicates that the value to be returned is STRINGB (referenced by 2/VAR), the longer string. This value is returned as the value of the whole #DF as control is returned to the point of the call.

As a different example, consider the following SIL code which calls the parser to parse the string at S/VAL using as the root production the syntax #DF with left-hand-side PRODUCTION:

```
S/VAL CVS/OP PRODUCTION/SYN STPRS/OP
```

The context-free grammar used by the parser is written using SEMANOL(76) syntactic #DF's. It is translated into SIL syntax statements as previously stated. One SIL syntax statement corresponds to each SEMANOL(76) syntactic #DF. Note that the set of legal SIL syntax statements is different from the set of SIL semantic statements (See Table 2). The parser uses the Jay Earley parsing algorithm, modified for SEMANOL(76), to compute a parse tree. The algorithm is documented in the listing and in several papers by Earley, so is not described here.

This concludes the discussion of INTERP which, indeed, passes control directly or indirectly to almost every other Executer subroutine at some time or other. The only thing that has not yet been mentioned is how INTERP halts. This can happen in one of two ways:

1. INTERP encounters MSTOP/OP or MERRS/OP in the SIL program control stream.
2. One of the operator subroutines detects an error in the SIL metaprogram. In this case an appropriate error message is printed at the terminal.

When either of these things happens, the Executer returns to Multics command level and waits for a further command.

Table I. SIL Syntax for Semantic Statements

#DF statement => <statement-name> < '/' > < 'SIL' > < gap > < '=' > < gap >
< list > < gap > < ';' > #.

#DF statement-name => < name > #U ['#CONTROL'] #.

#DF list => < '[' > < '[' > < gap > < element > < % < gap > < element > > > < gap >
< ']' > #.

#DF name => < any SEMANOL(76) name > #.

#DF element => < integer > < '/' > < 'VAR' >
=> < name > < '/' > < 'SIL' >
=> < name > < '/' > < 'SYN' >
=> < name > < '/' > < 'OP' >
=> < name > < '/' > < 'VAL' >
=> < constant >
=> < integer >
=> < '-' > < integer > #.

#DF constant => < any legal SEMANOL(76) string constant >
=> < '#B' > < gap > < any legal SEMANOL(76) bit-string constant >
=> < '#I' > < gap > < any legal SEMANOL(76) integer constant >
=> < '#TRUE' >
=> < '#FALSE' >
=> < '#UNDEFINED' >
=> < '#NULLSQ' > #.

#DF integer => < #DIGIT > < % < #DIGIT > > #.

#DF gap => < %1 < #SPACE, '[LF]' > > > #.

Table II. SIL Syntax for Syntactic Statements

```
#DF syntax-statement => <statement-name> <'/'> <'SYN'>
<gap> <'='> <gap> <'[['> <gap> <syntax-list> <gap>
<']> <gap> <';'> #.

#DF statement-name => name #.

#DF syntax-list => case
=> scat #.

#DF name => <any SEMANOL(76) name> #.

#DF case => <'('> <gap> <'CASE/OP'> <gap> <%1<<cat>
<gap>>> <')>'> #.

#DF scat => <'('> <gap> <'SCAT/OP'> <gap> <%1<<prim>
<gap>>> <')>'> #.

#DF cat => <'('> <gap> <%1<<prim> <gap>>> <')>'> #.

#DF prim      => set
               => union
               => setmin
               => kstar
               => kstar1
               => scanop
               => nterm
               => strlit #.

#DF set => <'('> <gap> <'SET/OP'> <gap> <%1<<strlit>
<gap> <'SETEND/OP'> <gap>>> <')>'> #.

#DF union => <'('> <gap> <'UNION/OP'> <gap> <%1<<alt>
<gap>>> <')>'> #.

#DF alt => <'('> <gap> <%1<<prim> <gap>>> <'UNEND/OP'>
<gap> <')>'> #.

#DF setmin => <'('> <gap> <'SETMIN/OP'> <gap> <%1<<prim>
<gap>>> <'SMEND/OP'> <gap> <'SMENDA/OP'> <gap> <%1<<strlit>
<gap>>> <')>'> #.

#DF kstar => <'('> <gap> <'KSTAR/OP'> <gap> <%1<<prim>
<gap>>> <'KEND/OP'> <gap> <')>'> #.

#DF kstar1 => <'('> <gap> <'KSTAR1/OP'> <gap> <%1<<prim>
<gap>>> <'KEND/OP'> <gap> <')>'> #.

#DF scanop => ['DIGIT/OP', 'SPACE/OP', 'ALPHA/OP', 'ASCII/OP',
'GAP/OP', 'NIL/OP', 'CAP/OP', 'DNUM/OP', 'LCASE/OP', 'NATNO/OP',
'EMPTY/OP'] #.
```

#DF nterm => <name> <' /SYN'> #.

#DF strlit => <any SEMANOL(76) string literal> #.

#DF gap => <%1<<#SPACE,'[LF]'\>>> #.

5. Logic Diagrams

The logic diagrams are presented on the following pages. There is an overall logic diagram and then one for each of the two major programs which constitute the SEMANOL(76) Interpreter.

6. Inputs

Program inputs are discussed separately for the two programs. Further discussion may be found in Section 9.

6.1 Translator Input

There is only one input file to the Translator, and that is an ASCII text file which contains the SEMANOL(76) source language program. The name of this file is a command parameter.

6.2 Executer Input

One to three input files are required to run the Executer. They are ASCII files like all other Multics text files. The first contains the SIL version of the SEMANOL(76) metaprogram segment to be run. It is required and its name is given in the `semanol` command. The second file is the ASCII text of the test case in the object language. It is optional and is referenced by the `run` command. The third file is the ASCII text of the input for the test run. It is also optional and is referenced by the `run` command.

7. Output

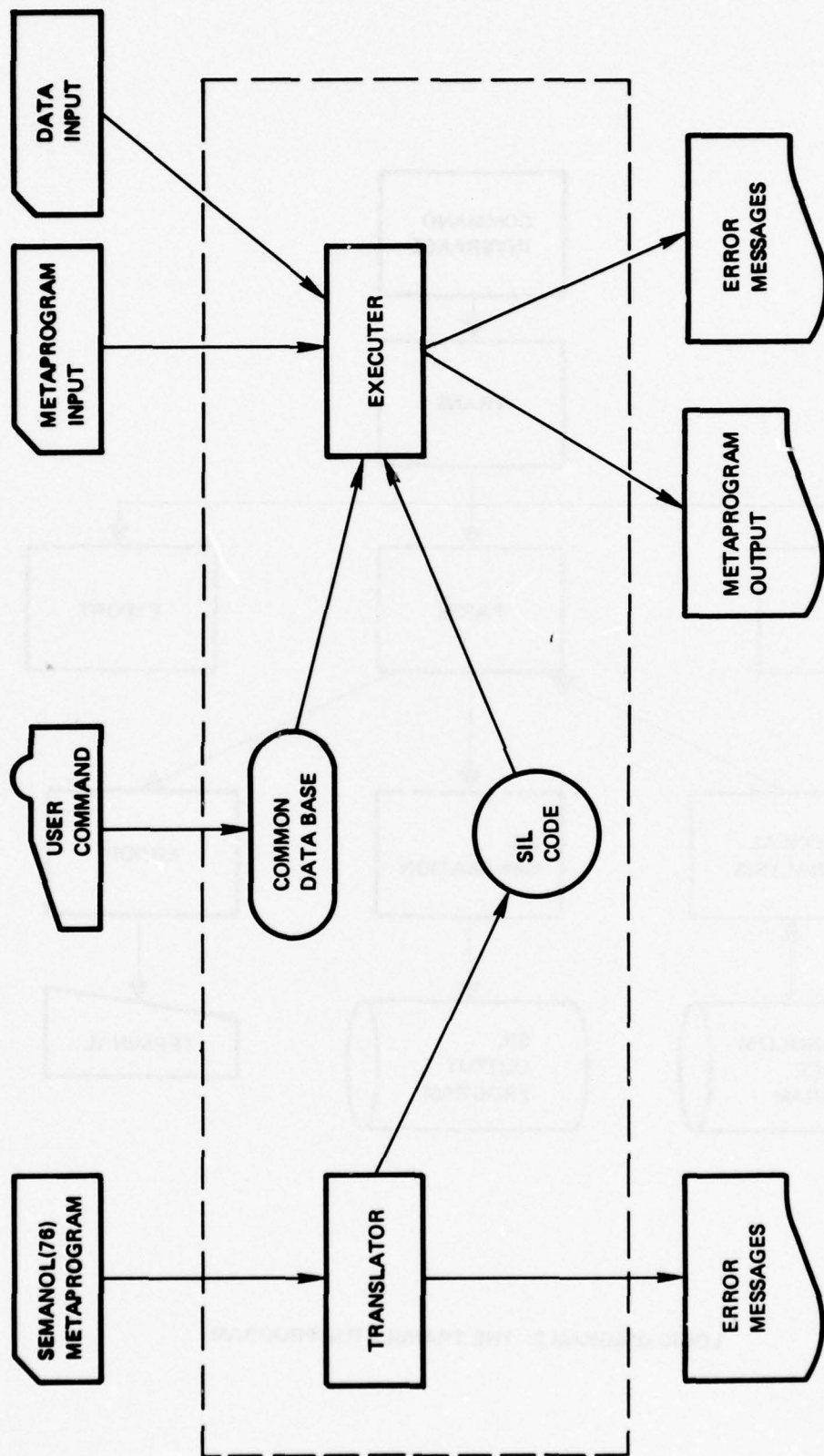
7.1 Translator Outputs

The major output of the Translator is the segment containing the generated SIL code. The name of this segment is a control command parameter. Other possible outputs are described in Section 9.1.

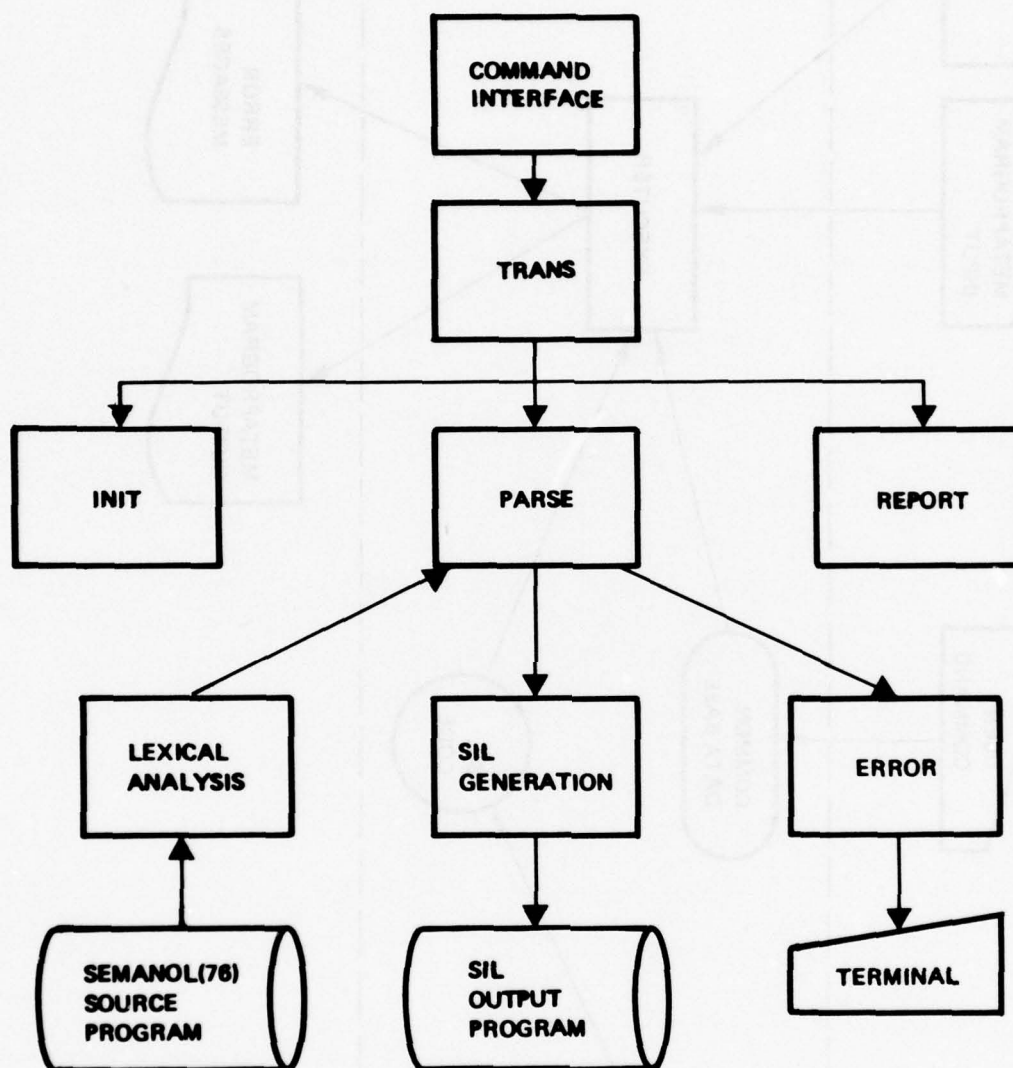
7.2 Executer Outputs

The main output file is the terminal. All object program output goes to this file. Other messages which go to this file are:

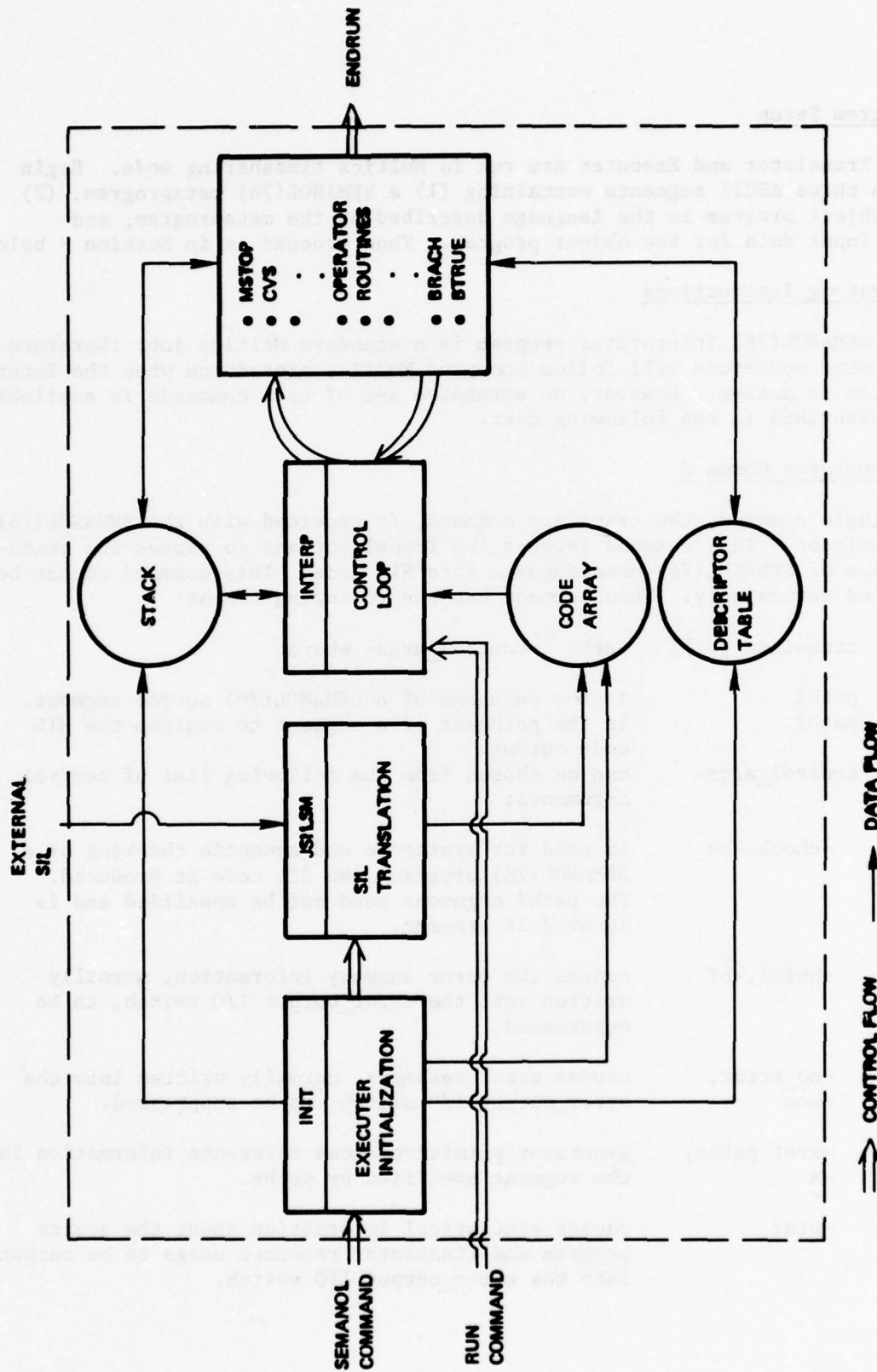
1. Any error messages output by Fortran.
2. Any error messages output by the Executer.
3. Messages indicating when the Executer garbage collector (GGC) and string compactor (SCOMP) are called and when they return.
4. #DF tracing messages when they are enabled.
5. A message "MSTOP CALLED" after each SEMANOL(76) program is run.



LOGIC DIAGRAM 1: THE SEMANOL(76) INTERPRETER SYSTEM



LOGIC DIAGRAM 2: THE TRANSLATOR PROGRAM



LOGIC DIAGRAM 3: THE EXECUTER PROGRAM

8. Program Setup

The Translator and Executer are run in Multics timesharing mode. Begin with three ASCII segments containing (1) a SEMANOL(76) metaprogram, (2) an object program in the language described by the metaprogram, and (3) input data for the object program. Then proceed as in Section 9 below.

9. Operating Instructions

The SEMANOL(76) Interpreter program is a standard Multics job; therefore, computer operators will follow accepted Multics procedures when the Interpreter is active. However, an extensive set of user commands is available as described in the following text.

9.1 Translator Command

A single command, the translate command, is provided with the SEMANOL(76) Translator. This command invokes the Translator and so causes the translation of SEMANOL(76) metalanguage into SIL code. This command cannot be called recursively. This command has the following format:

translate path1 path2 -control_args- where:

1. path1 is the pathname of a SEMANOL(76) source segment.
2. path2 is the pathname of a segment to contain the SIL code output.
3. control_args can be chosen from the following list of control arguments:
 - check,-ck is used for syntactic and semantic checking of a SEMANOL(76) program. No SIL code is produced. The path2 argument need not be specified and is ignored if present.
 - brief,-bf causes the error summary information, normally written into the error_output I/O switch, to be suppressed.
 - no_error,-noe causes error messages, normally written into the error_output I/O switch, to be suppressed.
 - xref pathx, -x generates primitive cross reference information in the segment specified by pathx.
 - stat causes statistical information about the source program and translator resource usage to be output into the error_output I/O switch.

-incremental -inc	allows incremental translation of partial SEMANOL(76) metaprograms. If no keyword is present in the input segment, then it is assumed to contain only semantic definitions. If no input segment is specified, the default path "[process_directory] >incremental_source" is used. If no output segment is specified, the default path "[process_directory] >incremental_sil" is used. The symbol table segments are assumed to be correct and are not reset. This means that use of the incremental option must follow a non-incremental translation performed since the last login or new_proc.
-debug -db	causes the debug program to be called as the last action after a Translator error. This is a testing feature, and is not intended for general use.

Invoking the translator without control arguments produces a SIL file, error messages and an error summary.

Error Diagnostics

The SEMANOL(76) Translator outputs four classes of errors.

1. Warning only. Compilation continues without ill effect. The messages in this class begin with the string "*W*".
2. Lexical errors. Compilation continues with the offending text converted to a unique illegal token. The messages in this class begin with the string "*L*".
3. Syntactic and semantic errors. Compilation continues but no SIL code will be generated for the #DF containing the error. The messages in this class begin with the string "*S*".
4. Compiler errors. Compilation is aborted. The output file is in an undefined state. The messages in this class begin with the string "*C*".

Error messages are written into the error_output I/O switch as they occur. An example of an error message follows.

#CONTEXT-FREE-SYNTAX:

#DF: syntaxo

5

=> #.

↑

S expected syntactic-expression after =>

The first line is the section in which the error occurred. It will be one of the keywords:

DECLARE-GLOBAL
 DECLARE-SYNTACTIC-COMPONENT
 CONTEXT-FREE-SYNTAX
 SEMANTIC-DEFINITIONS
 CONTROL-COMMANDS

The second line is the DF name in which the error occurred (e.g., syntax).
 The third line is the line number of the source in which the error occurred.
 The fourth line is the text of the line in which the error occurred. The
 fifth line is an indication of where in the line the error was detected.
 The sixth line is a descriptive error message. The first four lines are
 not repeated for additional error messages referring to the same source
 line.

9.2 Executer Commands

The Executer is that part of the SEMANOL(76) Interpreter which actually
 executes SEMANOL(76) metaprograms. Each Executer command is a separate
 program called from MULTICS command level. The programs communicate
 through FORTRAN COMMON blocks which are initialized by the `semanol` command.
 Note that the Translator is a separate program which communicates with the
 Executer commands only through SIL files. Some sample Executer commands
 follow:

```

semanol mini_basic.sil
run mini_basic.prog mini_basic.data
  
```

These commands assume (as do all of the examples that follow) that (1)
`mini_basic.sil` is an ASCII segment containing the Translator SIL output
 from translating a SEMANOL(76) metaprogram description of the demonstration
 language `mini_basic`, (2) `mini_basic.prog` is an ASCII segment containing a
 sample `mini_basic` program, and (3) `mini_basic.data` is an ASCII segment
 containing input data for the sample `mini_basic` program.

The first step prior to using the Executer is to establish links to the
 21 Executer commands. After this, the first command executed must be the
`semanol` command. The command

```
semanol mini_basic.sil
```

accomplishes several things:

- (1) It initializes the Executer and the COMMON blocks which provide
communication between the commands.
- (2) It loads all of the #DFs on the ASCII text segment `mini_basic.sil`.
(Presumably this segment contains the SIL output of a previous
`translate` command.)

The `semanol` command may produce output lines of the form

```
scomp called
scomp returns
```

or

```
ggc called
ggc returns
```

indicating that the string compactor or garbage collector was called. These routines are required to reclaim unused internal memory and take about 10 CPU seconds for the string compactor and 15 CPU seconds for the garbage collector to run. The messages are printed so that the user will know when his processing time is spent doing these overhead functions. Note that once the `semanol` command is executed, it need not be executed again during the current process.

The `run` command is used to actually begin execution of a SEMANOL(76) meta-program. Assume, for example, that the `semanol` command has just been executed, loading from `mini_basic.sil` the SIL corresponding to a SEMANOL(76) description of `mini_basic` (a simple demonstration language). Assume, also, that a `mini_basic` test program exists on file `mini_basic.prog` and that its input exists on file `mini_basic.data`. Then the program can be run with the specified input by typing

```
run mini_basic.prog mini_basic.data
```

The command can be executed again for an additional run, perhaps with different data.

Suppose the following `mini_basic` program is on file `mini_basic.prog`:

```
10 INPUT I
20 IF I >= 2 THEN 50
25 GOSUB 60
30 LET I = I + 1
40 GOTO 20
50 STOP
60 PRINT I
70 RETURN
80 END
```

(Each line is assumed to be followed by a line-feed.) And suppose 0 followed by a line-feed is on file `mini_basic.data`. Then the run output will look like the following:

```

run mini_basic.prog mini_basic.data

? 0

0

1

mstop called

in #CONTROL at location 54: level 1

STOP

r...

```

The first line is typed by the user and initiates the run of the sample mini_basic program and its data. The second line "?0" is output by the SEMANOL(76) metaprogram to indicate that 0 has been input by the INPUT I statement on line 10. The next two lines "0" and "1" are the output from the PRINT I statement on line 60. The next three lines are output by the Executer to indicate that a normal termination has occurred. The last line is the MULTICS ready message.

As with the semanol command, there are other possible outputs. Again

```

scomp called
scomp returns

```

indicates that a string compaction is taking place and

```

ggc called
ggc returns

```

indicates that a garbage collection is taking place. Either of these messages may occur during other commands, also. In the example run above, a #STOP was executed as indicated by the message "mstop called." In some other run an error message will be printed out in the form

```

#error executed
in error at location 12: level 5

```

The first line is the error message itself. The second line indicates (as in the normal termination in the example above) the location of the error (#DF name and relative code list location) and the level of the #DF stack at the time of error. Here #error was executed in a #DF named error at relative location 12 and #DF stack level 5. This is the standard form for any error message occurring during execution.

There are 19 Executer commands other than `semanol` and `run`. All must follow the `semanol` command and all are discussed in the following pages. Some general Executer command concepts follow:

- Whatever applies to #DFs, also applies to #PROC-DFs. The term "#DF" is used throughout.
- An error detected in a multiple argument command cancels processing in later arguments.
- A loaded #DF is one whose corresponding SIL has been read by a previous `semanol` or `load` command. An unloaded #DF is one whose SIL has not been so read.
- All commands terminate by printing `STOP` followed by the MULTICS ready message.
- An error message, a string compaction message, or a garbage collection message may appear at any time.

Break Commands

Associated with each loaded #DF is a break flag which may be on or off. If the break flag is on for a given #DF, execution is suspended whenever that #DF is called. The user may want to define some auxiliary #DFs which print important intermediate results. Then, when a break occurs, he can execute these #DFs using the `executedf` command. After a break has occurred, the user can continue execution with the `continue` command. A soft escape is available using the `interrupt` command. The available break commands are described below.

brlst

`brlst`

The `brlst` command lists the names of all loaded #DFs which have their break flag on.

No error can occur.

broff

`broff dfname1 ... dfnameN`

`dfnameI` the name of a loaded #DF

The `broff` command turns the break flag off for each #DF named in its argument list.

An error is signalled if there is no argument or if one of the arguments names an unloaded #DF.

bron

`bron dfname1 ... dfnameN`

`dfnameI` the name of a loaded #DF

The `bron` command turns the break flag on for each #DF named in its argument list.

An error is signalled if there is no argument or if one of the arguments names an unloaded #DF.

continue

`continue`

The `continue` command continues (resumes) execution after a break or error has suspended execution.

An error occurs if it is not legal to continue (e.g. because a `run` command was never executed).

interrupt

`interrupt`

The `interrupt` command sets a flag so that the Executer will break at the next #DF called.

No error can occur, but the command should only be used as explained below.

The use of the `interrupt` command is different from that of other commands. It is used to simulate a soft escape, i.e. a break set on the fly, from a running metaprogram. Simply typing the MULTICS escape may leave the Executer data structures in a compromised condition. The `interrupt` command allows a break to occur at a safe place.

Assume that the `run` command has been typed and that a metaprogram is in the midst of executing. To safely stop it the user should do the following:

- (1) Hit the MULTICS escape key. This returns the user to MULTICS command level and leaves the Executer in an unknown state.
- (2) Type the `interrupt` command. This executer command sets an interrupt flag in the COMMON communication area and then returns to MULTICS command level.

- (3) Type the MULTICS start command. This allows the Executer to continue at the escape point in (1) above. The Executer will then break at the next #DF called, leaving the system uncompromised.
- (4) At this point, the user can execute any command that he would normally execute after a break, e.g. he can continue.

Following is a sample session at the terminal which illustrates the various breaking commands. The user types the lines followed by a *, the computer types all other lines. Note that "r..." represents the MULTICS ready message and [escape] represents a user-typed escape:

```
semanol mini_basic.sil *
STOP
r ...

brlst *
STOP
r ...

bron simple-successor *
STOP
r ...

brlst *
simple-successor
STOP
r ...

run mini_basic.prog mini_basic.data *
? 0
break at simple-successor
in statement-successor-of at location 75: level 2
STOP
r ...

continue *
break at simple-successor
in if-then-successor at location 28: level 3
STOP
r ...

broff simple-successor *
STOP
r ...

brlst *
STOP
r ...
```

```

continue *
0
[escape]*
QUIT
r ...

interrupt *
r ...

start *
interrupt
break at sequence-of-executable-statements-in
in simple-successor at location 27: level 3
STOP
r ...

continue *
1
mstop called
in #CONTROL at location 54: level 1
STOP
r ...

```

Syntactic Component Commands

Associated with each loaded #DF is a syntactic component flag. In the default case, this flag is on if the #DF is declared as a #SYNTACTIC-COMPONENT in the SEMANOL(76) metaprogram, and the flag is off otherwise. The user may wish to incrementally override these declarations for one reason or another, and that is the purpose of the syntactic component commands. These commands are described below.

sclst

sclst

The sclst command lists the names of all loaded #DFs which have their syntactic component flag on.

No error can occur.

scoff

scoff dfname1 ... dfnameN

dfnameI the name of a loaded #DF

The scoff command turns the syntactic component flag off for each #DF named in its argument list.

An error occurs if there is no argument or if one of the arguments names an unloaded #DF.

scon

scon dfname1 ... dfnameN

dfnameI the name of a loaded #DF

The scon command turns the syntactic component flag on for each #DF named in its argument list.

An error is signalled if there is no argument or if one of the arguments names an unloaded #DF.

Note that in most cases scon should be used only if a run command (as opposed to a continue command) is to start execution.

Trace Commands

Associated with each loaded #DF is a trace flag which may have one of four possible values. By judiciously setting trace flags on various #DFs, the user can selectively trace desired portions of his SEMANOL(76) metaprogram's execution. The default trace flag value is trcneu. If all trace flags are set to this value, no tracing occurs.

When a #DF is called, a determination is made as to whether that #DF is to be traced. If a #DF is traced, a message indicating its name and the level number of the call is printed at both the call and the return. The returned value is also printed at the return. The following table indicates whether any given #DF is traced:

#DF trace flag value	Action
TRCON	Trace the #DF, independent of its caller.
TRCOFF	Do not trace the #DF, independent of its caller.
TRCTEM	Trace the #DF, independent of its caller. (See TRCNEU for difference from TRCON)
TRCNEU	Trace the #DF if its caller was traced and its caller did not have trace flag value TRCTEM. Otherwise, do not trace the #DF.

tracefile

tracefile file.trace

file.trace optional pathname of a segment to receive the trace output

Trace output is normally (by default) sent to the terminal. The tracefile command directs subsequent trace output to the specified segment. If no segment is specified, subsequent trace output is again sent to the terminal. Note that to print a trace output segment, the user must first type the MULTICS adjust_bit_count command, giving the pathname of the segment as argument.

trlst

trlst

The trlst command lists the names and local trace flag values of all loaded #DFs which do not have local trace flag value trcneu. Possible listed trace flag values are trcon, trcoff, and trctem.

No error can occur.

trneu

trneu dfname1...dfnameN

dfnameI the name of a loaded #DF

The trneu command sets the local trace flag for each #DF named in its argument list back to the default value trcneu.

An error occurs if there is no argument or if one of the arguments names an unloaded #DF.

troff

troff dfname1...dfnameN

dfnameI the name of a loaded #DF

The troff command sets the local trace flag for each #DF named in its argument list to trcoff.

An error occurs if there is no argument or if one of the arguments names an unloaded #DF.

tron

tron dfname1...dfnameN

dfnameI the name of a loaded #DF

The tron command sets the local trace flag for each #DF named in its argument list to tron.

An error occurs if there is no argument or if one of the arguments names an unloaded #DF.

trtem

trtem dfname1...dfnameN

dfnameI the name of a loaded #DF

The trtem command sets the trace flag for each #DF named in its argument list to trtem.

An error occurs if there is no argument or if one of the arguments names an unloaded #DF.

Following is a sample session at the terminal which illustrates the various tracing commands. The user types the lines followed by a *, the computer types all other lines. Note that "r..." represents the MULTICS ready message. Also, note that #CONTROL is the name used to refer to the #CONTROL-COMMANDS section:

```
semanol mini_basic.sil *
STOP
r...
```

```
tron #CONTROL *
STOP
r...
```

```
trlst *
#CONTROL tron
STOP
r...
```

```
run mini_basic.prog mini_basic.data *
0 call of #CONTROL
1 call of is-syntactically-valid
:
```

All #DFs are traced in the above example.

Miscellaneous Commands

The remaining commands described below are neither break commands, nor syntactic component commands, nor trace commands. Note that this section further expands on the `semanol` and `run` commands introduced previously.

calst

`calst`

The `calst` command prints the current state of the `#DF` call stack, one `#DF` name per line. It is used after an error or break.

No error can be signalled by `calst`.

executedf

`executedf dfname`

`dfname` the name of a loaded `#DF`

The `executedf` command begins execution by calling the named `#DF` with no arguments. A `run` command may have been previously executed, but that is not required.

An error occurs if there is no argument or if the argument names an unloaded `#DF`.

load

`load file.sil`

`file.sil` mandatory pathname of an ASCII segment containing SIL output produced by the Translator

The `load` command loads a SIL file, making its `#DFs` ready to run. The command can be used to incrementally add or update SIL code. The last loaded version of any `#DF` is the one used when a new `#DF` call occurs. Note that the break, trace, and syntactic component flags for any incrementally updated `#DF` are reset to their default values (i.e., break to off, trace to `trcneu`, and syntactic component to on or off, depending on declarations in the translated `SEMANOL(76)` metaprogram).

An error occurs if an attempt is made to load a `#DF` currently on the `#DF` call stack (i.e., in execution). An error also occurs if the load argument is missing.

prcl

prcl dfname1 dfname2 ... dfnameN

dfname1 the name of a loaded #DF

The prcl command prints the internal form of the SIL code for each #DF named in its argument list. It is of use only to those familiar with this internal form.

An error occurs if there is no argument or if one of the arguments names an unloaded #DF.

reset

reset

The reset command resets the Executer to a state all ready to begin execution. All #DFs currently in execution are unstacked, the same as in the run command. Execution does not begin. The purpose of reset is to restore the Executer to a point at which a #DF previously in execution can be reloaded using the load command.

No error can occur.

run

run file.prog file.data

file.prog optional pathname of an ASCII segment containing the string to be returned by #GIVEN-PROGRAM during this execution

file.data optional pathname of an ASCII segment containing the string to be returned by #INPUT during this execution

The run command first resets the Executer #DF call stack to level 0 (i.e., to empty) and assigns #UNDEFINED to all global variables and names in the #ASSIGN-LATEST-VALUE space. It then begins running the previously loaded #DF. The first code executed is that corresponding to the #CONTROL-COMMANDS section.

Many different execution errors can occur as explained previously. Also, an error will be signalled if either optional pathname specifies a non-accessible file.

semanol

semanol file.sil

file.sil optional pathname of an ASCII segment containing SIL output produced by the Translator

The **semanol** command initializes the Executer and its associated COMMON areas. It must be executed prior to the execution of any other Executer command. Optionally, the command loads a SIL file, making its #DFs ready to run.

An error occurs if the optional pathname specifies a non-accessible or empty file, or if the contents of the specified file contains illegal SIL.